一、Linux系统相关

1. Linux系统文件格式：7/8种

普通文件：-

目录文件：d 目录

字符设备文件：c 即串行端口的接口设备，例如键盘、鼠标等等

块设备文件：b 就是存储数据以供系统存取的接口设备，简单而言就是硬盘。

软链接：l 是一种特殊文件，指向一个真实存在的文件链接，类似于Windows下的快捷方式，链接文件的不同，又可分为硬链接文件和符号链接文件。

管道文件：p 是一种很特殊的文件，主要用于不同进程的信息传递。

套接字：s 这类文件通常用在网络数据连接。可以启动一个程序来监听客户端的要求，客户端就可以通过套接字来进行数据通信。

未知文件。
占磁盘空间：普通文件、目录、软链接
其他的都是伪文件，不占磁盘空间

2.文件权限rwx

用户对文件的权限

r—4　　w—2　　x—1

rwx|r-x|r-x　所有者u、同组用户g、其他o

使用chown 一次修改所有者和所属组：

3.Linux软链接、硬链接

软链接

ln -s xiao xiao.s 用相对路径给xiao文件或者目录创建一个软链接xiao.s

软链接相当于快捷方式，有大小，大小为访问路径，相对路径创建的软链接大小为4字节

相对路径创建的软链接不可以像windows快捷方式那样剪贴到别的路径

ln -s ./xiao xiao.soft 绝对路径创建软链接xiao.soft

绝对路径创建的软链接大小为6字节

绝对路径创建的软链接可以像windows快捷方式那样剪贴到别的路径（用什么路径创建的就可以在什么路径中访问，建议用/home/xiaodexin/桌面/main/xiao，这个路径大小为36字节）

ln -s /home/xiaodexin/桌面/main/xiao xiao.soft

注意：为保证软链接可以随意搬移，最好用绝对路径创建软连接

硬链接

ln xiao.txt xiao.h 硬连接不加-s

大小等于源文件大小

对创建的硬链接文件进行修改，与之相链接的文件（包括源文件以及其他硬链接文件）也会发生改变，两者是同步的

大概相当于指针的思想 有相同的Inode

4.GCC编译

gcc编译：

```
4步骤：  预处理、编译、汇编、连接。

-I：  指定头文件所在目录位置（头文件与源文件不在同一路径下）。

-c：  只做预处理、编译、汇编。得到  二进制  文件！！！（看不懂）

-g：  编译时添加调试语句。  主要支持  gdb  调试。

-Wall：  显示所有警告信息。

-E：      生成预处理文件.i

-D：  向程序中“动态”注册宏定义。     #define NAME VALUE
（相当于嵌入式中在设置中直接定义宏定义，即编译时注册宏定义）
-l：指定动态库名
-L：指定动态库路径
```

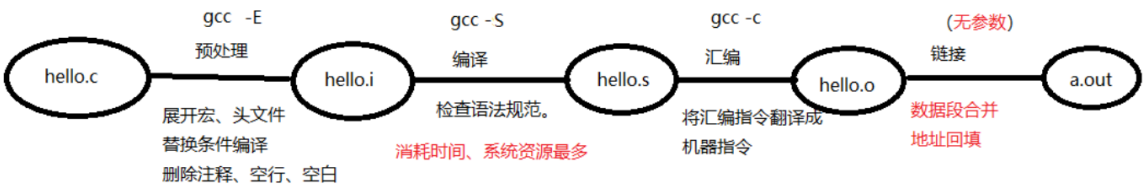
预编译（预处理）：gcc -E hello.c -o hello.i　　去掉注释，导入头文件，展开宏定义

编译：gcc -S hello.i -o hello.s　　高级语言代码变为汇编代码

汇编：gcc -c hello.s -o hello.o　　汇编代码变为目标代码（二进制010101）

链接：gcc hello.o -o hello.out　　目标代码变为可执行文件

gcc 编译可以执行程序 4 步骤：预处理、编译、汇编、链接



5.阻塞非阻塞、同步异步

IO同步就是调用网络IO接口的时候(就是IO函数)，当数据准备好了以后，数据的读写是应用层自己去读写的，这个耗时的操作其实都是在应用程序中完成的，只有当应用程序把接收缓冲区中的数据搬到buf中搬完了，程序才能继续往下进行，这就是同步

异步要看操作系统是否提供相应的服务，提供异步的IO接口让你去调用，(同步是我们自己判断的)，在异步情况下，操作系统会帮助你把TCP接收缓冲区中的数据放到buf中，而应用程序在此期间可以去做其他的事情，，数据搬运好了以后会发信号通知应用程序，应用程序再去处理这个数据。

1.阻塞/非阻塞、同步/异步(网络IO)

> 典型的一次IO的两个阶段是什么？数据就绪 和 数据读写

以买机票为例：
同步相当于，你买完机票，要跑到机场取票，然后再回来，这个取票的过程是你自己完成的，花费的是你自己的时间。
异步的方式相当于你在购票的时候，机场也提供了邮寄的服务，于是你留了邮寄的地址，买完票以后由机场的工作人员给你邮寄过去，到了你家楼下打电话通知你，你就可以直接拿到机票。
同步就是你需要自己去取，异步是别人帮你做好了，然后通知你。
同步因为要自己搬数据，消耗应用程序的时间，而异步是内核搬的，不需要耗费应用程序的时间，它的效率一定没有异步高，但是同步编程难度低，调用读或接收函数，有数据到了就去处理。

数据就绪：根据系统IO操作的就绪状态

阻塞和非阻塞是指缓冲区没有数据到达时，
我是在卡在这里挂起进程，还是返回做其他事情

- 阻塞
- 非阻塞

数据读写：根据应用程序和内核的交互方式

同步异步是指从内核拷贝数据到用户区是应用程序自己去拷贝的还是交由内核去帮助我们拷贝的

- 同步
- 异步

是说read、recv、write、send这些操作IO的函数都是同步IO，都需要应用程序自己来操作数据

陈硕：在处理 IO 的时候，阻塞和非阻塞都是同步 IO，只有使用了特殊的 API 才是异步 IO。

常见的问题：IO多路复用是同步的

同步	IO multiplexing (select/poll/epoll)	
	阻塞	非阻塞

异步	Linux	Windows	.NET
	AIO	IOCP	BeginInvoke/EndInvoke

IO同步和异步

> 一个典型的网络IO接口调用，分为两个阶段，分别是“数据就绪” 和 “数据读写”，数据就绪阶段分为阻塞和非阻塞，表现得结果就是，阻塞当前线程或是直接返回。
>
> 同步表示A向B请求调用一个网络IO接口时（或者调用某个业务逻辑API接口时），数据的读写都是由请求方A自己来完成的（不管是阻塞还是非阻塞）；异步表示A向B请求调用一个网络IO接口时（或者调用某个业务逻辑API接口时），向B传入请求的事件以及事件发生时通知的方式，A就可以处理其它逻辑了，当B监听到事件处理完成后，会用事先约定好的通知方式，通知A处理结果。

所谓同步和异步就是在于数据到底是由谁去进行操作的，数据是我自己去搬运读写的就是同步，而由别人写好以后通过一种方式告诉我的就是异步

- 同步阻塞
- 同步非阻塞
- 异步阻塞
- 异步非阻塞

6.进程、线程控制原语对比

线程控制原语	进程控制原语
pthread_create()	fork()
pthread_self()	getpid();
pthread_exit()	exit(); / return
pthread_join()	wait()/waitpid()
pthread_cancel()	kill()
pthread_detach()	

7.Linux内核态和用户态

内核态：也叫**内核空间**，是内核进程/线程所在的区域。**主要负责运行系统、硬件交互。**每个进程的 4G 地址空间中，最高 1G 都是一样的，即内核空间。换句话说就是， 最高 1G 的内核空间是被所有进程共享的！

用户态：也叫**用户空间**，是用户进程/线程所在的区域。**主要用于执行用户程序。**每个进程的 4G 地址空间中的第0~3G空间为用户空间。

内核态和用户态的区别
内核态：运行的代码不受任何限制，CPU可以执行任何指令。

用户态：运行的代码需要受到CPU的很多检查，不能直接访问内核数据和程序，也就是说不可以像内核态线程一样访问任何有效地址。

在 CPU 的所有指令中，有些指令是非常危险的，如果错用，将导致系统崩溃，比如清内存、设置时钟等。如果允许所有的程序都可以使用这些指令，那么系统崩溃的概率将大大增加。在内核状态下，进程运行在内核地址空间中，此时 CPU 可以执行任何指令。运行的

代码也不受任何的限制，可以自由地访问任何有效地址，也可以直接进行端口的访问。
在用户状态下，进程运行在用户地址空间中，被执行的代码要受到 CPU 的诸多检查。**所以，区分内核空间和用户空间本质上是要提高操作系统的稳定性及可用性。**

操作系统在执行用户程序时，主要工作在用户态，只有在其执行没有权限完成的任务时才会切换到内核态。也就是**当进程运行在内核空间时就处于内核态，而进程运行在用户空间时则处于用户态。**

各种函数

open/close函数

```
SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);
```

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
```

- 常用参数

O_RDONLY、O_WRONLY、O_RDWR

O_APPEND、O_CREAT、O_EXCL、 O_TRUNC、 O_NONBLOCK

创建文件时，指定文件访问权限。权限同时受 umask 影响。结论为：

文件权限 = mode & ~umask

使用头文件：<fcntl.h>

- open 常见错误 ：

1. 打开文件不存在
2. 以写方式打开只读文件(打开文件没有对应权限)
3. 以只写方式打开目录

read/write函数

read函数:

ssize_t read(int fd, void *buf, size_t count);

参数:

fd：文件描述符

buf：存数据的缓冲区

count：缓冲区大小

返回值:

0：读到文件末尾。

成功； > 0 读到的字节数。

失败： -1， 设置 errno

-1： 并且 errno = EAGIN 或 EWOULDBLOCK, 说明不是read失败，而是read在以非阻塞方式读一个设备文件（网络文件），并且文件无数据。

write函数：

ssize_t write(int fd, const void *buf, size_t count);

参数：

fd：文件描述符

buf：待写出数据的缓冲区

count：数据大小

返回值

成功； 写入的字节数。

失败： -1， 设置 errno

read/write这块，每次写一个字节，会疯狂进行内核态和用户态的切换，所以非常耗时**。

fgetc/fputc，有个缓冲区（标库IO函数自带用户级缓冲区），4096，所以它并不是一个字节一个字节地写，内核和用户切换就比较少

预读入，缓输出机制。

所以系统函数并不是一定比库函数牛逼，能使用库函数的地方就使用库函数。

标准IO函数自带用户缓冲区，系统调用无用户级缓冲。系统缓冲区是都有的。

lseek函数

移动文件内部“指针”（偏移位置）

```
off_t lseek(int fd, off_t offset, int whence);

参数：
    fd：文件描述符
    offset：  偏移量
    whence：起始偏移位置：  SEEK_SET/SEEK_CUR/SEEK_END

返回值：

    成功：较起始位置偏移量
    失败：-1 errno

应用场景：
    1. 文件的“读”、“写”使用同一偏移位置。

    2. 使用lseek获取文件大小（直接将lseek设置到END，返回值就是文件大小）

    3. 使用lseek拓展文件大小：要想使文件大小真正拓展，必须引起IO操作。

        使用 truncate 函数，直接拓展文件。  int ret = truncate("dict.cp", 250);
```

fcntl函数

fcntl：改变一个已经打开了的文件的访问属性

```
int (int fd, int cmd, ...)
fd    文件描述符
cmd   命令，决定了后续参数个数
```

```
//设置文件为非阻塞
int flgs = fcntl(fd,  F_GETFL);
flgs |= O_NONBLOCK;
fcntl(fd,  F_SETFL, flgs);

重点掌握两个参数的使用，  F_GETFL，F_SETFL
获取文件状态：  F_GETFL
设置文件状态：  F_SETFL

★位图思想（类似于嵌入式中的寄存器思想）
```

终端文件默认是阻塞读的，这里用fcntl将其更改为非阻塞读

stat、lstat函数

```
头文件：
#include <unistd.h>
#include <sys/stat.h>

int stat(const char *path, struct stat *buf);//会穿透符号链接
int lstat(const char *path, struct stat *buf);//不会穿透符号链接

获取文件属性（从inode结构体中获取）

参数：
    path：  文件路径
    buf：（传出参数）  存放文件属性。

返回值：
    成功：  0
    失败：  -1 errno

S_ISREG(buf.st_mode)
```

```
S_ISDIR(buf.st_mode)
...
获取文件大小：  buf.st_size
获取文件类型：  buf.st_mode
获取文件权限：  buf.st_mode
符号穿透：stat会。lstat不会。
```

目录操作函数opendir、closedir、readdir

目录操作函数：

```
#include <sys/types>
#include <dirent.h>
DIR *opendir(char *name); //打开一个目录，成功返回目录指针，失败返回NULL，DIR*：目录结构体指针，相当于文件操作中的FILE*

int closedir(DIR *dp); //关闭一个目录，成功返回0失败返回-1，设置errno

struct dirent *readdir(DIR * dp);//读取目录,返回目录项dentry，成功返回结构体dentry，每次返回一个，到最后一个后返回NULL，失败返回NULL，设置errno

struct dirent {
        inode
        char d_name[256];
}
```

重定向（dup和dup2）

`int dup(int oldfd);` 文件描述符复制。

oldfd: 已有文件描述符

返回：新文件描述符，这个描述符和oldfd指向相同内容。

★ `int dup2(int oldfd, int newfd);`

文件描述符复制，oldfd拷贝给newfd。返回newfd

失败返回-1，设置errno

fork函数

```
 pid_t fork(void)
//创建子进程。父子进程各自返回。父进程返回子进程pid， 子进程返回 0。 失败父进程返回-1，不返回子进程。通过判断返回值来区别是父进程还是子进程
```

fork之后，是父进程先执行还是子进程先执行不确定，取决于内核所使用的调度算法。

fork之前的代码，父子进程都有，但是只有父进程执行了，子进程没有执行，fork之后的代码，父子进程都有机会执行。

两个函数：

pid_t getpid()　获取当前进程id

pid_t getppid()　获取当前进程的父进程id

getpid函数

```
#include <unistd.h>

pid_t getpid(void);
```

获取自己的进程id

getppid函数

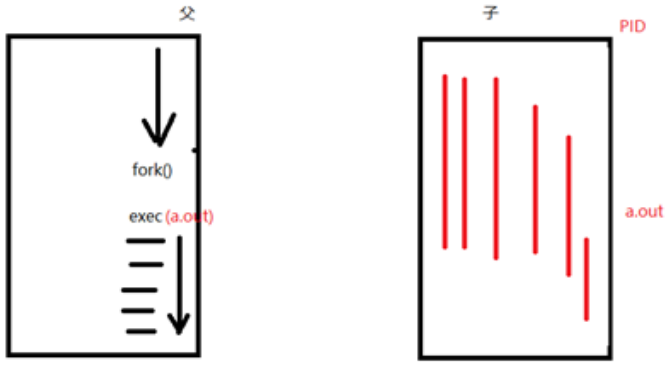
```
#include <unistd.h>

pid_t getppid(void);
```

获取父进程id

exec函数族

fork创建子进程后执行的是和父进程相同的程序（但有可能是不同的代码分支），子进程往往要调用一种exec函数以执行另一个程序。**当进程调用一种exec函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行。**调用exec并不创建新进程，所以调用前后该进程的id并未改变。

将当前进程的.text、.data替换为所要加载程序的.text、.data，然后让进程从新的.text第一条指令开始执行，但进程ID不变，换核不换壳。



exec函数族：

使进程执行某一程序。成功无返回值，失败返回 -1，设置errno

头文件：#include <unistd.h>

```
int execlp(const char *file, const char *arg, ...);
```

借助 PATH 环境变量找寻待执行程序（常用于调用系统执行程序）

参1： 程序名

参2： argv0

参3： argv1

...： argvN

哨兵：NULL

```
int execl(const char *path, const char *arg, ...);
```

自己指定待执行程序路径。

```
int execvp();
```

```
ps ajx --> pid ppid gid sid
```

execlp函数

```
    int execlp(const char *file, const char *arg,
...);      借助 PATH 环境变量找寻待执行程序，常用来调用系统程
序


        参1:  程序名

        参2:  argv0

        参3:  argv1c++

        ...:  argvN

        哨兵：NULL
例：
        execlp("ls", "ls", "-l", "-h", NULL);
        execlp("date", "date", NULL);
```

★execl函数

```
    int execl(const char *path, const char *arg,
...);       自己指定待执行c++程序路径。
例：
    execl("./a.out", "./a.out", NULL);
    execl("/bin/ls", "ls", "-l", NULL); //也可以是系统
程序，第一个参数是路径
```

execvp函数

加载一个进程，使用自定义环境变量env

```
    int execvp(const char *file, const char
*argv[]);
例：
    char *argv[] = {"ls", "-l", "-h", NULL};
    execvp("ls", argv);
```

实际上与ececlp一样，只是参数形式不一样了。

wait/waitpid函数（回收子进程）

★wait函数

一个进程终止时会关闭所有文件描述符，释放在用户空间分配的内存，但它的PCB还保留着，内核在其中保存了一些信息：如果是正常终止则保存着退出状态，如果是异常终止则保存着导致该进程终止的信号是哪个。这个进程的父进程可以调用wait或者waitpid获取这些信息，然后彻底清除掉这个进程。我们知道一个进程的退出状态可以在shell中用特殊变量$?查看，因为shell是它的父进程，当它终止时，shell调用wait或者waitpid得到它的退出状态，同时彻底清除掉这个进程。

wait函数： 回收子进程退出资源， 阻塞回收任意一个。

```
pid_t wait(int *status)

  头文件：#include <sys/wait.h>
  参数：（传出） 回收进程的状态。
  返回值：成功： 回收进程的pid
        失败： -1， errno
  函数作用1： 阻塞等待子进程退出
  函数作用2： 清理子进程残留在内核的 pcb 资源
  函数作用3： 通过传出参数，得到子进程结束状态
获取子进程正常终止值：
```

```
    WIFEXITED(status) --》 为真 --》调用
WEXITSTATUS(status) --》 得到 子进程 退出值。

获取导致子进程异常终止信号：
    WIFSIGNALED(status) --》 为真 --》调用
WTERMSIG(status) --》 得到 导致子进程异常终止的信号编号。
获取导致子进程挂起信号：
    WIFSTOPPED(status) --》 得到 导致子进程挂起的信号编
号。
获取导致子进程恢复的信号：
    WIFCONTINUED(status) --》 得到 导致子进程恢复的信号编
号。
```

★waitpid函数

waitpid函数： 指定某一个进程进行回收。可以设置非阻塞。

waitpid(-1, &status, 0) == wait(&status);

```
  pid_t waitpid(pid_t pid, int *status, int options)

  参数：
    pid：指定回收某一个子进程pid
        >0： 待回收的子进程pid
        -1：任意子进程
        0：同组的所有子进程。
        <-1:指定进程组内的任意子进程
    status：（传出） 回收进程的状态。
    options：WNOHANG 指定回收方式为，非阻塞。
  返回值：
    > 0 ： 表成功回收的子进程 pid
    0 ： 函数调用时， 参3 指定了WNOHANG， 并且，没有子进程结
束。
    -1: 失败。errno
```

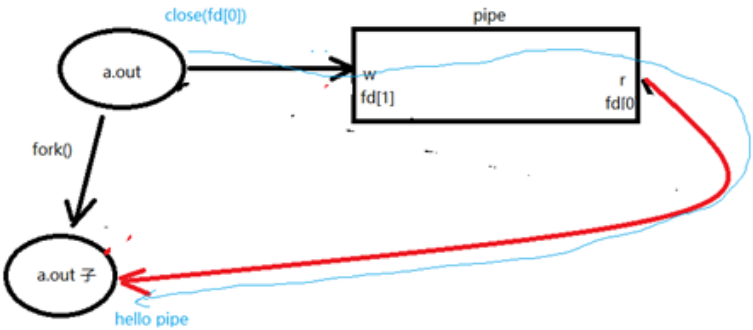
总结：一次wait/waitpid函数调用，只能回收一个子进程，且只能回收子进程。上一个例子，父进程产生了5个子进程，wait会随机回收一个，捡到哪个算哪个。若想回收多个，用while循环依次回收

★pipe函数

pipe函数： **创建，并打开管道。**

```
int pipe(int fd[2]);

 参数：  fd[0]: 读端。
         fd[1]: 写端。

 返回值：  成功：  0     失败：  -1 errno
```

管道通信原理：



先在父进程创建管道，此时管道的读端和写端都是指向父进程，然后fork一个子进程，子进程中也有相同的管道，其读端和写端都指向子进程，此时去掉父进程的读端和子进程的写端，就建立了一个父进程写、子进程读的管道。

管道的读写行为

读管道read：

1）管道有数据，read返回实际读到的字节数。

2） 管道无数据：

①无写端，read返回0 （类似读到文件尾）

②有写端，read阻塞等待。

写管道write：

1） 无读端， 异常终止。 （SIGPIPE导致的）

2） 有读端：

① 管道已满， write阻塞等待 （少见）

② 管道未满， write返回写出的字节个数。

mkfifo函数创建有名管道

命令： mkfifo+文件名

函数：

```
#include<sys/types>
#include<sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
//成功返回0，失败返回-1，设置errno
```

mmap函数创建内存映射

```
void *mmap(void *addr, size_t length, int prot, int
flags, int fd, off_t offset);     创建共享内存映射

  参数：
```

addr：　指定映射区的首地址。通常传NULL，表示让系统自动分配

length：共享内存映射区的大小。（<= 文件的实际大小）

prot：　共享内存映射区的读写属性。PROT_READ、PROT_WRITE、PROT_READ|PROT_WRITE

flags：　标注共享内存的共享属性。MAP_SHARED、MAP_PRIVATE

flags里面的shared意思是修改会反映到磁盘 private表示修改不反映到磁盘上

fd：用于创建共享内存映射区的那个文件的 文件描述符。

offset：默认0，表示映射文件全部。偏移位置。需是 4k 的整数倍。

返回值：

成功：映射区的首地址。

失败：MAP_FAILED (void*(-1))， 设置errno

```
int munmap(void *addr, size_t length);  释放映射区。
```

addr：mmap 的返回值

length：大小

成功返回0，失败返回-1

signal函数

注册一个信号捕捉函数

```
#include <signal.h>
typedef void (*sighandler_t)(int);  //定义一个名为sighandler_t的类型，其指针指向返回值为空，参数为int的函数。（函数指针）
sighandler_t signal(int signum, sighandler_t handler);//回调handler
//该函数由ANSI定义，由于历史原因在不同的版本的Unix和Linux中可能有不同的行为，因此应该尽量避免使用signal函数，取而代之的是sigaction函数
```

sigaction函数

注册一个信号捕捉函数

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
            signum:信号编号
            act：新的处理状态
            oldact：旧有的处理状态，传出参数
 struct sigaction {
        void     (*sa_handler)(int);//跟signal函数差不多，回调
        void     (*sa_sigaction)(int, siginfo_t *, void *);//一般不用
        sigset_t   sa_mask; //在信号捕捉函数运行期间代替mask，只工作于信号捕捉函数运行期间
        int        sa_flags;//一些设置参数
        void     (*sa_restorer)(void);  //废弃
};


Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
    sigaction(): _POSIX_C_SOURCE
```

```
    siginfo_t: _POSIX_C_SOURCE >= 199309L
```

信号捕捉特性

```
1. 捕捉函数执行期间，信号屏蔽字 由 mask --> sa_mask ，捕捉函数执行结束。 恢复回mask

2. 捕捉函数执行期间，本信号自动被屏蔽(sa_flgs = 0).

3. 捕捉函数执行期间，被屏蔽信号多次发送，解除屏蔽后只处理一次!
```

pthread_self函数

```
#include<pthread.h>
pthread_t pthread_self(void);    获取线程id。 线程id是在进程地址空间内部，用来标识线程身份的id号，与LWP线程号不一样。
        返回值：本线程id

检查出错返回：    线程中。
    fprintf(stderr, "xxx error: %s\n", strerror(ret));
```

pthread_create函数

```
int pthread_create(pthread_t *tid, const
pthread_attr_t *attr, void *(*start_rountn)(void *),
void *arg); 创建子线程。
        参1：传出参数，表新创建的子线程的线程id
        参2：线程属性。传NULL表使用默认属性。
        参3：子线程回调函数。创建成功，ptherad_create函数
返回时，该函数会被自动调用，返回值void*，参数void*。
        参4：参3的参数。没有的话，传NULL
        返回值：成功：0
                失败：errno
    注意：编译的时候要加-pthread表示引入线程库
```

pthread_exit函数

['eksɪt]

```
void pthread_exit(void *retval);  退出当前线程。
        retval：退出值。 无退出值时，NULL

各种形式退出线程的区别：
        exit(); 退出当前进程。
        return: 返回到调用者那里去。
        pthread_exit(): 退出当前线程。
```

pthread_join函数

```
int pthread_join(pthread_t thread, void **retval);
//阻塞等待线程退出，获取线程退出状态。 回收线程。对应进程中的
waitpid()
        thread: 待回收的线程id
        retval：传出参数。 回收的那个线程的退出值。
            线程异常结束，值为 -1。
        返回值：成功：0
              失败：errno
```

注意：兄弟线程间可以回收，即创建一个线程回收其他线程，而兄弟进程间不能回收！

pthread_detach函数

实现线程分离。只有线程有这个机制，进程没有。

线程分离状态：指定该状态，线程主动与主控线程断开关系。线程结束后，其退出状态不由其他线程获取，而直接自己自动释放。**网络、多线服务器常用。**

进程若有这个机制，则将不会产生僵尸进程。僵尸进程的产生主要由于进程死后，大部分资源被释放，一点残留资源仍存于系统中，导致内核认为该进程仍存在。

也可使用pthread_create函数参2(线程属性)来设置线程分离。

```
int pthread_detach(pthread_t thread);          设置线程分离
        thread: 待分离的线程id
        返回值：成功：0
                失败：errno
```

pthread_cancel函数

[ˈkænsl]

```
int pthread_cancel(pthread_t thread);          杀死（取消）一个线程。    需要到达取消点（保存点）
        thread: 待杀死的线程id
        返回值：成功：0
                失败：errno
```

如果，子线程没有到达取消点， 那么 pthread_cancel 无效。

我们可以在程序中，手动添加一个取消点。使用

```
pthread_testcancel();
```

成功被 pthread_cancel() 杀死的线程，返回 -1.使用 pthead_join 回收。

mutex相关函数

主要应用函数：

pthread_mutex_init	初始化函数
pthread_mutex_destory	销毁函数
pthread_mutex_lock	加锁函数
pthread_mutex_trylock	尝试加锁函数
pthread_mutex_unlock	解锁函数

以上5个函数的返回值都是：成功返回0，失败返回错误号

pthread_mutex_t 类型，其本质是一个结构体。为简化理解，应用时可忽略其实现细节，简单当成整数看待

pthread_mutex_t mutex；变量mutex只有两种取值：0,1

条件变量cond相关函数

```
pthread_cond_init();     //初始化
pthread_cond_destory(); //销毁
pthread_cond_wait();     //等待，阻塞等待某一条件变量满足
pthread_cond_timedwait();//等待，等待某一条件满足，设置超时时间
pthread_cond_signal();   //通知唤醒，一次唤醒阻塞在条件变量上的（至少）一个线程
```

```
pthread_cond_broadcast();//广播通知，一次唤醒阻塞在条件变量上的所有线程
```

以上6个函数的返回值都是成功返回0，失败直接返回错误号

```
pthread_cond_t 类型，用于定义条件变量
pthread_cond_t cond;
  初始化条件变量：
   1. pthread_cond_init(&cond, NULL);          动态初始化。
   2. pthread_cond_t cond = PTHREAD_COND_INITIALIZER;   静态初始化。
```

二.进程

1.进程三种状态

(其实是5种)

运行态：进程占用cpu，正在cpu上运行

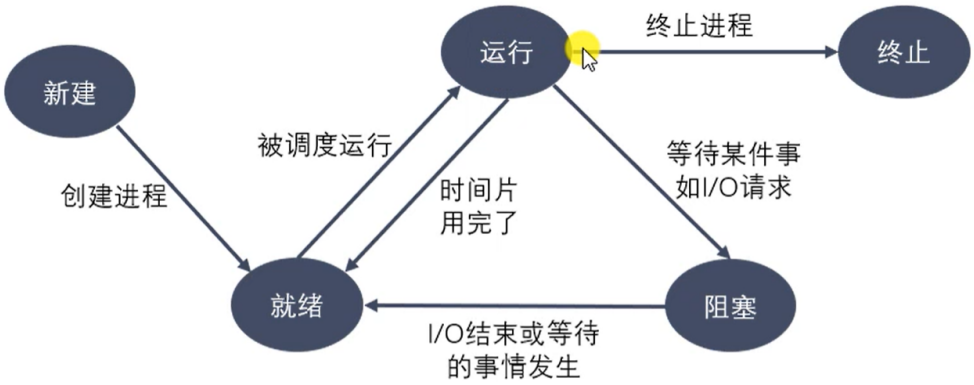
就绪态：已经具备运行条件，但由于没有空闲cpu，而暂时不能运行

阻塞态（挂起态）：因等待某一事件而暂时不能运行

除此之外还有创建态和终止态

创建态（初始态）：进程正在被创建，操作系统为进程分配资源、初始化PCB

终止态：进程正在从系统中撤销，操作系统回收进程拥有的资源



2.进程间通信方式

进程通信是指进程之间的信息交换，进程的通信方式有：

(1)**管道**，管道是连接读写进程的一个共享文件，实质就是内存中开辟的一个缓冲区（内核缓冲区），管道只能半双工通信分为有名管道和无名管道

无名管道只能用在有亲缘关系的进程间,比如父子进程(因为要共享文件描述符)。

有名管道(FIFO)允许无亲缘关系的进程间进行通信(通过文件路径)。

一个进程创建了一个管道,并调用fork创建自己的一个子进程后,父进程关闭读管道端,子进程关闭写管道端,这样就提供了两个进程之间数据流动的一种方式。

无名管道：优点：简单方便；缺点：1）局限于单向通信2）只能创建在它的进程以及其有亲缘关系的进程之间;3）缓冲区有限；
有名管道：优点：可以实现任意关系的进程间的通信；缺点：1）它会长期存于系统中，使用不当容易出错；2）缓冲区有限

(2)**信号量**
信号量可以看作是一个整型计数器，可以表示资源的数量，主要是用来实现**进程间的互斥和同步。**

控制信号量的方式有两种原子操作，一个是P操作，这个操作会把信号量减1，如果相减后信号量<0，则表明资源被占用，进程需阻塞等待；相减后信号量>=0，则表明还有资源可用，进程可正常继续执行。另一个是V操作，这个操作会把信号量加1，相加后信号量<=0,表明当前有阻塞中的进程，于是唤醒该进程，相加后信号量>0,表示当期没有阻塞中的进程。

优点：可以同步进程；缺点：信号量有限

(3)**消息队列**

消息队列是消息的链表,存放在内核中并由消息队列标识符标识.Linux允许不同进程将格式化的数据流以消息队列形式发送给任意进程。消息队列克服了信号传递信息少,管道只能承载无格式字节流和缓冲区大小受限的问题.
优点：可以实现任意进程间的通信，并通过系统调用函数来实现消息发送和接收的同步，不用考虑同步问题，方便；缺点：存在用户态与内核态之间的数据拷贝开销。

(4)**共享内存**：(不经过内核，它是最快的方式)

共享内存就是拿出一块虚拟地址空间，映射到能被其他进程所访问的物理内存中,这段共享内存由一个进程创建,多个进程都可以访问，节省了很多拷贝操作，提高了进程间通信的速度.

共享内存是最快的IPC(进程间通信)方式.(它往往与其他通信机制,如信号量,配合使用,来实现进程间的同步与通信.)

优点：无须复制，快，信息量大；缺点：通信是通过将无法实现共享空间缓冲区直接附加到进程的虚拟地址空间中来实现的，因此进程间的读写操作的同步问题；

(5)**信号**：信号是一种比较复杂的通信方式,对于异常情况下的工作模式，就需要用信号的方式通知进程某个事件已经发生.

(6)**socket套接字**：可用于不同及其间的进程通信
优点：1）传输数据为字节级，传输数据可自定义，数据量小效率高；2）传输数据时间短，性能高；3) 适合于客户端和服务器端之间信息实时交互；4) 可以加密,数据安全性强
缺点：1) 需对传输的数据进行解析，转化成应用级的数据不同主机的进程间使用socket进行通信(上述方法是相同主机间的进程通信)。

3.Linux中进程调度算法

在进程的生命周期中，当进程从一个运行状态变化到另外一个运行状态时，就会触发一次调度。

1. 先来先服务调度算法（First Come First Seved, FCFS）：每次调度都是从就绪队列中选择最先进来进入队列的进程，然后一直运行它，直到进程退出或被阻塞，才会继续从队列中选择第一个进程继续运行。当一个长作业(作业就是一个任务进程)先运行了，那么后面的短作业等待时间会很长，所以这个调度算法不利于短作业。
2. 最短作业优先调度算法（Shortest Job First, SJF）：每次都是从就绪队列中优先选择运行时间最短的进程(作业)来运行，有助于提高系统的吞吐量。
3. 高响应比优先调度算法：该算法主要是为了权衡短作业和长作业，每次进行进程调度时，先计算响应比优先级，然后把响应比最高的进程投入运行。响应比计算公式：优先级=(进程等待时间+要求服务时间)/要求服务时间.
4. 时间片轮转法：每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。

4.孤儿、僵尸、守护进程

孤儿进程：一个进程退出后，它的一个或多个子进程还在运行，那么这些子进程将成为孤儿进程。孤儿进程最终会被init进程(进程号为1)所收养。(父进程先于子进程终止)

僵尸进程：指一个进程使用fork函数创建子进程以后，子进程退出了，而父进程并没有调用wait()或waitpid()函数清理子进程的相关信息。那么子进程依然存在于系统中，占用系统资源，这种进程称为僵尸进程。在此期间，kill对其无效。

僵尸进程的解决办法：使用ps aux | grep Z命令查看进程表中的僵尸进程信息，然后使用kill杀死它的父进程，让它变成孤儿进程，然后被系统init进程收养并清理。

所有的子进程没被回收都会变成僵尸进程吗?

不是，init进程不会，任何一个子进程(init除外)在exit()之后，并非马上就消失掉，而是留下一个称为僵尸进程的数据结构(它占用一点内存资源)，等待父进程处理。这是每个子进程在结束时都要经过的阶段。如果子进程在exit()之后，父进程没有来得及处理，这时用ps命令就能看到子进程的状态是“Z”。

守护进程（daemon进程）：守护进程是一个在后台运行并且不受任何终端控制的进程,(守护进程是有意把父进程结束，被1号init进程收养，init进程是内核启动的第一个用户级进程)。

★守护进程创建过程：

1、在父进程中执行fork命令，并退出父进程，这样这个子进程会被init进程托管。

2、子进程中调用setsid()函数创建一个新的会话，并担任该会话组的组长，这样可以使该子进程脱离终端的控制，让这个进程单独成一个进程组，摆脱父进程的影响。

3、因为子进程继承了父进程的工作目录，所以在子进程中调用chdir()函数，改变它的工作目录， 放在一个不可被卸载的目录下，防止目录被卸载。

```
int chdir(const char *path);
```

4、子进程中调用umask()函数重设文件权限掩码为0，也是因为子进程继承了之前父进程的掩码所以重设(文件权限掩码是指屏蔽掉文件权限中的对应位)

```
mode_t umask(mode_t mask);
```

022 -- 755 0345 --- 432 r---wx-w- 422

5、在子进程中关闭任何不需要的文件描述符，防止文件描述符串用

6、守护进程 业务逻辑。while（）

开始执行守护进程核心工作守护进程退出处理程序模型

5.pthread_detach()和pthread_join()

pthread_detach()是主线程和子线程分离，子线程结束后，它的资源会被系统自动回收。

pthread_joined()是子线程合入主线程，主线程阻塞等待子线程结束，然后回收子线程资源。

6. CAS操作如何保证原子性

CAS(compare and swap)

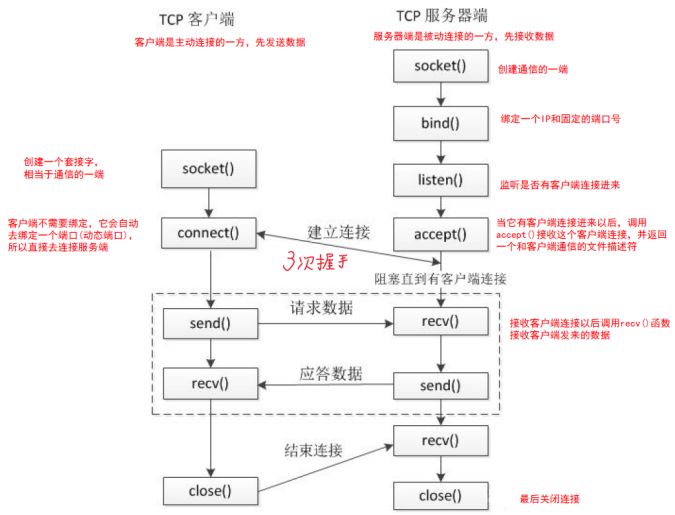
CAS(对比并交换）原理：CAS是一个原子指令，用于在多线程环境中实现同步。 它包括三个操作数：内存位置(V)、预期原值(A)、新值(B)。如果内存位置的值与预期原值相匹配，处理器会自动将该位置值更新为新值(说明没被其他线程修改，现在可改)，否则，不做任何操作。这个过程是作为单个原子操作完成的。原子性保证了这个

新值是最新的，这个新值在被读取至操作完成过程中不会被其它线程修改。

7. socket原理

(1)、请说一下socket通信流程/说一下socket通信原理/socket网络编程中用到哪些函数?
先解释下什么是socket套接字：socket是在应用层和传输层之间的一个抽象层，它本质上是编程接口(API)，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用来实现进程在网络中通信。
(TCP/IP只是一个协议栈，必须要具体实现，同时还要提供对外的操作接口（API），这就是Socket接口。通过Socket,我们才能使用TCP/IP协议。)

(2)通信流程



服务器端：
(1)服务器端调用socket()函数，根据IP地址(IPV4、IPV6)、套接字类型(数据报套接字、数据流套接字)、传输协议(TCP、UDP)来创建一个用于监听的套接字。
(2)服务器端创建完套接字以后，调用bind()函数将创建出来的套接字与服务器一个有效的IP地址和端口号进行绑定，这样流经该IP地址和端口的数据才会交给套接字处理。
(3)服务器端调用listen()函数让套接字处于监听状态，并设置可以同时和服务器建立连接的客户端的上限数。

(4)服务器端调用accept()函数阻塞等待，直到接收到客户端的连接请求才解除阻塞，accept()函数会返回一个新的用于和客户端通信的套接字。
(5)接下来就是接收到客户端连接以后，和客户端进行通信，调用write()和read()函数进行数据的读写。

客户端：
(1)客户端也调用socket()函数创建一个用于通信的套接字。
(2)客户端调用connect()函数，根据服务器IP地址和端口号去尝试连接服务器。调用connect函数将激发TCP的三次握手过程，而且仅在连接建立成功或出错时返回。
(3)连接成功以后，客户端就可以和服务器通信，调用write()和read()函数进行数据的读写。

9.上下文切换(进程、线程)

各个进程之间是共享CPU资源的，在不同的时候操作系统会在CPU上切换不同的进程或线程，这就是进程/线程的上下文切换。

上下文切换的步骤：

1、将前一个CPU的上下文（也就是CPU寄存器和程序计数器里边的内容）保存起来

2、然后加载新任务的上下文到寄存器和程序计数器；

3、最后跳转到程序计数器所指的新位置，运行新任务。

·被保存起来的上下文会存储到**系统内核**中，等待任务重新调度执行时再次加载进来。

·CPU的上下文切换分三种：**进程上下文切换、线程上下文切换、中断上下文切换。**

·进程上下文切换：进程是由内核管理和调度的，进程的切换只能发生在内核态。进程的上下文不但包括虚拟内存、栈、全局变量等用户空间资源，还包括内核堆栈、寄存器等内核空间状态。所以，进程的上下文切换比系统调用多一个步骤：保存当前进程的内核状态和CPU寄存器之前，**先把该进程的虚拟内存、用户栈等保存起来；加载下一个进程的内核态后，还需要刷新进程的虚拟内存和用户栈。**保存上下文和恢复上下文需要内核在CPU上运行才能完成。

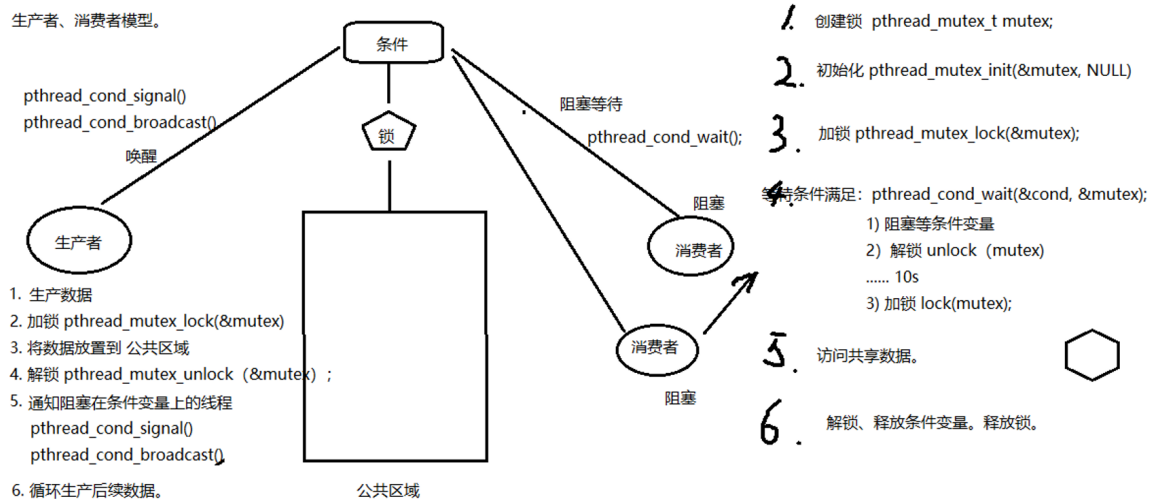
·**线程上下文切换：**共享相同的虚拟内存和全局变量等资源不需要修改。线程自己的私有数据，如栈和寄存器等，上下文切换时需要保存。

·**中断上下文切换：**中断上下文，需要保存内核态中断服务程序执行所必需的状态，包括**CPU**寄存器、内核堆栈、硬件中断参数等。

10.进程地址空间如何保证独立

操作系统通过给每个进程分配一套独立的虚拟地址来将各个进程所使用的地址隔离开来,每个进程都不能访问物理地址，通过OS提供的机制，再将不同进程的虚拟地址和不同内存的物理地址映射起来。

11.概述一个消费者-生产者模型是怎样的



生产者-消费者模型就是创建一个生产者线程以及多个消费者线程，同时创建一个公共区域/任务队列来存放生产者生产的数据，这个任务队列是互斥资源，无论是消费者取数据任务还是生产者村数据任务都需要进行加锁解锁操作。创建一个条件变量，当任务队列中没有数据任务的时候，所有的消费者线程都阻塞在这个条件变量上；当生产者线程产生数据任务并通过加锁解锁操作将其放入任务队列后，会唤醒阻塞在条件变量上的一个消费者线程，消费者线程被唤醒后通过加锁解锁操作到任务队列中取走数据任务进行处理。

（可以结合项目的线程池一起说）

12.并行与并发

并发是指在操作系统中，一个时间段中有多个进程都处于已启动运行到运行完毕之间的状态。但，任一时刻点上仍只有一个进程在运行。宏观上可以看作是多个进程在同时运行。

并行则是真正指的是多个进程在同时进行，并行需要硬件支持如多核CPU、分布式操作系统等等。

13.进程共享（父子进程）

父子进程相同：

刚fork后。 data段、text段、堆、栈、环境变量、全局变量、宿主目录位置、进程工作目录位置、信号处理方式

父子进程不同：

进程id、返回值、各自的父进程、进程创建时间、闹钟（定时器）、未决信号集

似乎看上去是子进程复制了父进程0-3G用户空间内容，以及父进程的PCB，但pid不同，实际上每fork一个子进程都要将父进程0-3G地址空间完全拷贝一份然后再映射到物理内存吗？当然不是，父子进程之间遵循**读时共享写时复制**的原则，这样设计，无论子进程执行父进程的逻辑还是执行自己的逻辑都能节省内存开销。

父子进程共享：

读时共享、写时复制。————————— 全局变量。

真正共享：

1. 文件描述符 2. mmap映射区（进程间通信详解）。

三.线程

1.线程间通信方式

因为同一进程下的多个线程是共享内存地址空间的，所以他们可以通过共享全局变量的方式来进行通信，但是要注意线程间的同步问题，来保证共享数据的一致性。

线程概念：LWP：light weight process 轻量级进程，其本质仍为进程（在linux环境下）

进程：有独立的 进程地址空间。有独立的pcb。 分配资源的最小单位。

线程：**有独立的pcb。没有独立的进程地址空间。** 最小的执行单位。

ps -Lf 进程id ---> 线程号。LWP --》cpu 执行的最小单位。

2.线程同步原因

多个线程之间共享同一个进程的虚拟地址空间，同一进程下的多个线程会因为共享资源的竞争，导致数据错乱，比如多个线程同时读写同一个共享数据，就可能会发生冲突，所以引入线程同步机制，来解决共享数据的安全问题。

(线程没有单独的地址空间，但是每个线程有自己独立的一套寄存器和栈)

3.抽象锁

乐观锁和悲观锁是抽象概念的锁，不是系统自带的api，而是需要我们自己去实现，互斥锁、读写锁、自旋锁都是悲观锁。

乐观锁：乐观锁比较乐观，他认为多线程同时修改共享资源概率比较低，所以先修改共享资源，再验证这段时间内有没有发生冲突，如果没有其他线程修改资源，那么操作完成，如果发现有其他线程修改过这个资源，就放弃本次操作，乐观锁是无锁编程，

悲观锁：悲观锁比较悲观，他认为多线程同时修改共享资源的概率比较高，很容易出现冲突，所以访问共享资源前，先要上锁。

5.进程、线程、协程含义

进程是程序运行的实例，**是操作系统分配资源的基本单位。**

线程是轻量级的进程，**是操作系统调度的基本单位。**

协程是轻量级的线程，相当于对一个特殊函数的调用，它不是被操作系统管理的，而是被用户程序所控制。

协程最大的优势是执行效率高，因为它是用户程序来控制的，没有线程切换的开销

6.进程、线程、协程区别

进程和线程的区别

资源开销方面：因为进程具有一个完整的资源平台，所以在创建进程时，还需要一些资源管理信息(内存、文件这些信息)，而同一进程下的多个线程间是共享代码段、数据段以及文件资源的，所以线程的创建比进程更快，占用CPU的时间更少。同样线程的销毁也要比进程更快，因为线程释放的资源比进程少。另外，每个进程有独立的代码段、数据段和文件资源，而线程间共享这些内容，所以进程切换时的系统开销比线程更大。

影响关系方面：一个进程崩溃后，在保护模式下不会影响其他进程，但是一个线程挂掉，对应的进程都会挂掉了

数据传递方式不同：同一进程下的线程之间共享数据空间(全局变量、静态变量)，所以线程间传递数据简单，不需要经过内核，而进程间的数据相互独立，需要使用通信的方式来相互传递数据(包括，管道、消息队列、共享内存、信号、信号量、套接字)。

线程和协程的区别

协程不需要多线程的锁的机制，因为多个协程同属于一个线程中，不会出现同时写变量的冲突。

进程和线程都是同步机制，协程是异步机制

8.多进程与多线程适应场景

1、需要频繁创建和销毁的优先使用多线程。例如：web服务器，来一个建立一个线程，断了就销毁线程。要是用进程，创建和销毁的代价是很难承受的。

2、需要进行大量计算的优先使用多线程。大量计算会很消耗CPU，切换频繁，这种情况用线程最合适。例如：图像处理，算法处理。

3、进程有独立的地址空间，一个进程崩溃，在保护模式下不会影响其他进程。而线程没有单独的地址空间，多线程共用同一个进程的地址空间，一个线程死掉整个进程就死掉，所以多进程比多线程更安全

4、可以扩展到多机分布的用多进程，多核分布的用多线程。

9.多线程共享与不共享资源

同一进程下的多个线程在同一个虚拟地址空间，彼此之间共享文件描述符表、当前的工作目录、共享内存地址空间(包括其中的代码段、数据段以及文件资源)，不共享数据包括：线程ID、栈、寄存器。

10.多线程一定比单线程好吗

(举一个单线程比多线程好的例子)

不一定，因为多线程的上下文切换和创建线程都会消耗系统资源。

CPU密集型：主要特点是需要进行大量的计算，消耗CPU资源，对视频进行高清解码，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低。所以要最高效地利用CPU。计算密集型任务同时进行的数量应当等于CPU的核心数。

IO密集型：主要涉及网络、磁盘IO的任务都是IO密集型任务，这类任务特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成(因为IO的速度远低于CPU的速度)。对于IO密集型任务，任务越多，cpu效率越高，但也有一个限度

11.读写锁/互斥锁/自旋锁

见项目整理

四.内存分配

1.OS申请和管理内存

(1)申请

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：**brk和mmap**

brk是将进程数据段(.data)的最高地址指针向高处移动，通过这种方式扩大进程在运行时的堆大小。
mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）寻找一块空闲的虚拟内存，这样可以获得一块可以操作的堆内存。

一般分配的内存如果小于128k时，使用brk调用来获得虚拟内存，如果大于128k时使用mmap来获得虚拟内存。 进程先通过这两个系统调用获取虚拟内存，获得相应的虚拟地址，然后在访问这些虚拟地址的时候，通过缺页中断，让内核分配相应的物理内存,并建立虚拟内存和物理内存之间的映射关系，完成内存分配。

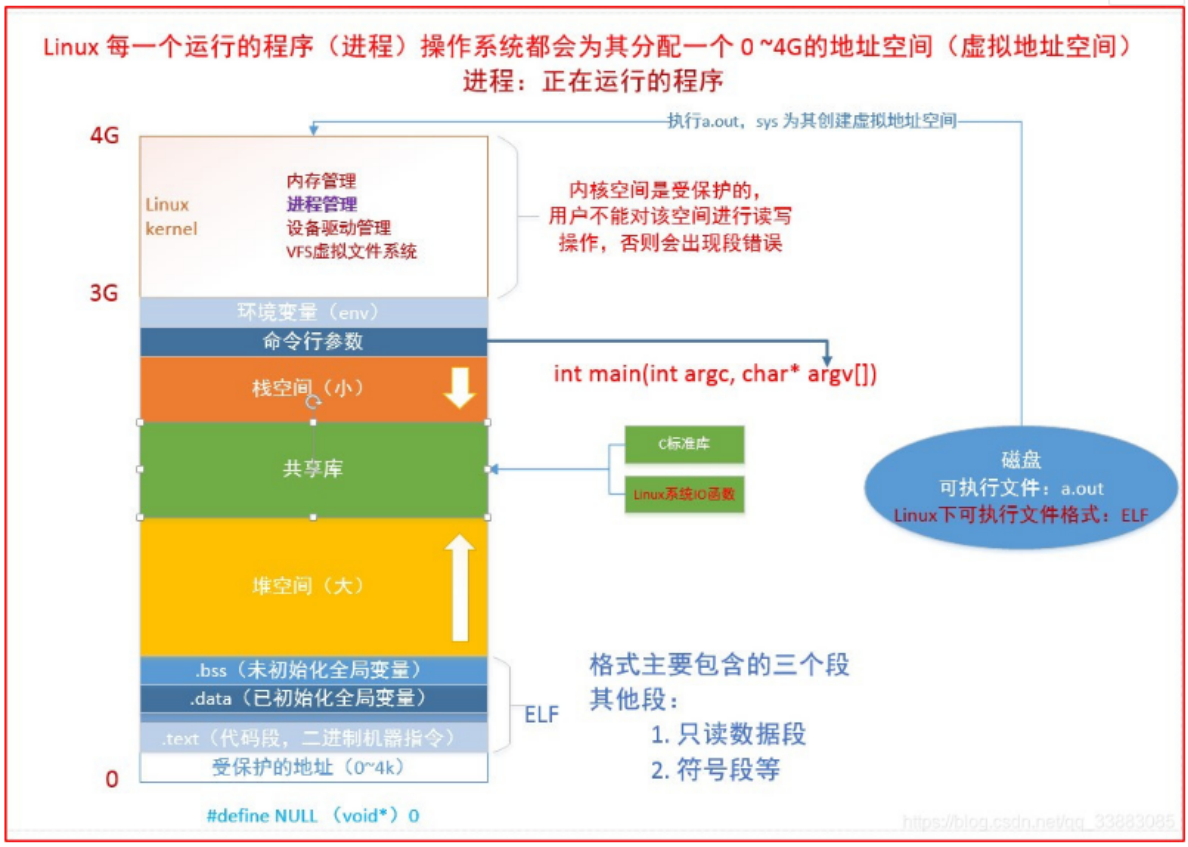
(2)管理

物理内存：物理内存有四个层次，分别是寄存器、高速缓存、主存、磁盘。

操作系统会对物理内存进行管理，有一个部分称为**内存管理器(memory manager)**，它的主要工作是有效的管理内存，记录哪些内存是正在使用的，在进程需要时分配内存以及在进程完成时回收内存。

虚拟内存：操作系统为每一个进程分配一个独立的地址空间，就是虚拟内存。虚拟内存与物理内存存在映射关系，通过页表寻址可以完成虚拟地址和物理地址的转换。

2.虚拟内存技术：



·虚存技术允许一个进程部分数据装入内存就可以运行，解决了进程地址空间隔离问题的一种存储管理技术。

·虚拟内存技术为每个进程提供私有的地址空间, 就好像它独占整个内存空间.所有进程共享同一物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。

·在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局，具体就是初始化进程控制表中内存相关的链表，实际上并不立即就把虚拟内存对应位置的程序数据和代码(比如.text .data段)拷贝到物理内存中，只是建立好虚拟内存和磁盘文 件之间的映射就好(叫做存储器映射)，等到运行到对应的程序时，才会通过缺页异常，来拷贝数据。

3.虚拟内存优缺点

优点：

（1）扩大地址空间。每个进程独占一个4G空间，真实物理内存没那么多。

（2）内存保护：防止不同进程对物理内存的争夺，可以对特定内存地址提供写保护，防止恶意篡改。

（3）可以实现内存共享，方便进程通信(两个进程映射同一块物理内存)。

（4）可以避免内存碎片，虽然物理内存可能不连续，但映射到虚拟内存上可以连续。

缺点：

（1）虚拟内存需要额外构建数据结构(虚存的分区)，占用空间。

（2）虚拟地址到物理地址的转换，增加了系统执行时间。

（3）页面的换入换出需要磁盘IO，比较耗时的

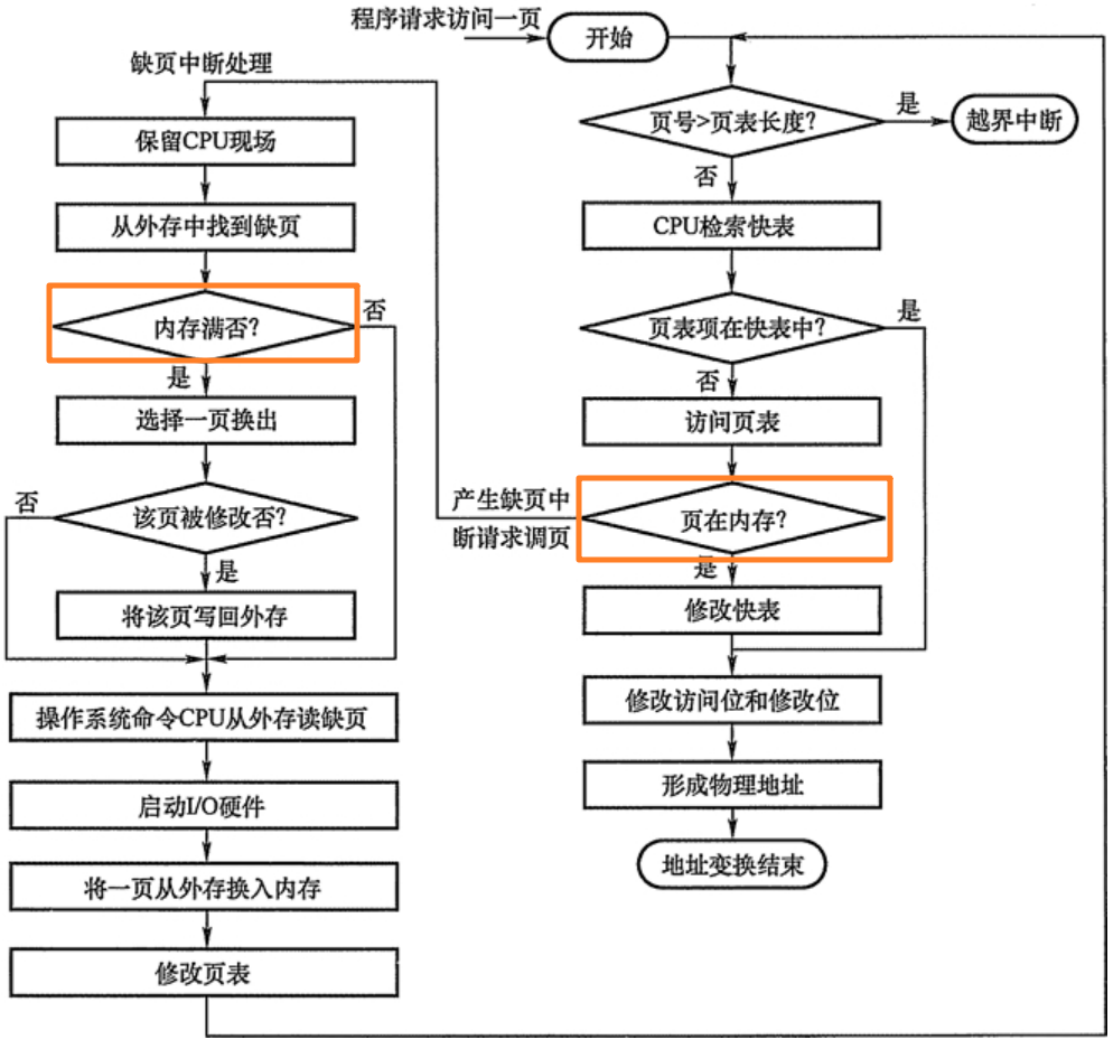
（4）一页如果只有一部分数据，浪费内存。

4.页面置换算法

简单回顾下虚拟内存技术，基于局部性原理来实现，总结起来就是两句话：

1. 在程序执行过程中，当 CPU 所需要的信息不在内存中的时候，由操作系统负责将所需信息从外存（磁盘）调入内存，然后继续执行程序
2. 如果调入内存的时候内存空间不够，由操作系统负责将内存中暂时用不到的信息换出到外存

整个请求调页的过程大概是这样的：



那么，**到底哪些页面该被从内存中换出来，哪些页面又该被从磁盘中调入内存呢**?

这就是『页面置换算法』干的事儿。

(1)最优页面置换算法(opt)

当一个缺页中断发生时，对于保存在内存当中的每一个逻辑页面，计算在它的下一次访问之前，还需要等待多长时间，从中选择等待时间最长的那个，作为被置换的页面。

缺点：这是理想情况，实际系统无法知道一个页面要等多长时间才会被再次访问。可用作其他算法的性能评价依据。

(2)先进先出置换算法（FIFO）

最简单的页面置换算法是先入先出（FIFO）法。总是选择在主存中停留时间最长（即最老）的一页置换，即先进入内存的页，先退出内存。因为最早调入内存的页，其不再被使用的可能性比刚调入内存的可能性大。建立一个FIFO队列，收容所有在内存中的页。被置换页面总是在队列头上进行。当一个页面被放入内存时，就把它插在队尾上。

优点：实现简单
缺点：产生缺页次数相对比较多

(3)最近最少使用（LRU）算法

根据程序**局部性原理**，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。使用一个栈，新页面或者命中的页面移动到栈底，每次替换栈顶的缓存页面。

优点：LRU算法对热点数据命中率是很高的。
缺点：需要寄存器和栈的硬件支持

(4)最不经常访问(LFU)算法

置换最近一段时间访问次数最少的页面。如果数据过去被访问多次，那么将来被访问的频率也更高。每个数据块一个引用计数，所有数据块按照引用计数排序，具有相同引用 计数的数据块则按照时间排序。每次淘汰队尾数据块。

缺点：排序需要一定的开销。

5.虚拟内存到物理内存

对于一个内存地址转换，其实就是这样三个步骤：

1. 把虚拟内存地址，切分成页号和对应页内偏移量的组合；
2. 从页表里面，查询出虚拟页号对应的物理内存块号，内存块号*内存块的大小得到物理内存的起始地址；

3. 物理内存的起始地址，加上前面的偏移量，就得到了虚拟内存地址对应的物理内存地址。

6.页表

为了知道进程的每个页面在内存中的实际位置，操作系统就为每个进程建立一张虚拟内存到物理内存的映射表，就是页表。

一个进程对应一张页表，页表是记录进程页面和实际存放的内存块之间的对应关系，进程的每一页对应一个页表项，每个页表项由“页号”和“内存块号”（页码号+页框号）组成。(内存块号*内存块的大小=该号内存的起始地址)

设立页表的原因：不可能将每一个虚拟内存的字节都对应到物理内存的地址上。进行分页，这样可以减小虚拟内存页对应物理内存页的映射表大小。

7.TLB(快表机制)

每次访问一个逻辑地址，都需要查询内存中的页表，由局部性原理可知，可能连续很多次查到的都是同一个页表项，所以引入快表机制(TLB)，它是一种访问速度比内存快很多的高速缓冲存储器，用来存放当前访问的若干页表项，以加速地址变换的过程。与此对应，内存中的页表常称为慢表。

引入快表后，地址变换过程

1.CPU给出逻辑地址，由某个硬件算得页号、页内偏移量，将页号与快表中的所有页号进行比较。

2.如果在快表中找到匹配的页号，说明要访问的页表项在快表中有副本，则直接从快表中取出该页表项中对应的内存块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。因此，若快表命中，则访问某个逻辑地址仅需一次访问即可。 3. 如果没有找到匹配的页号，则需要访问内存中的页表(慢表)(这是第一次访存)，找到对应页

表项，得到页面存放的内存块号，再将内存块号与页内偏移量拼接形成物理地址，访问该物理地址对应的内存单元(这是第二次访存)。因此，若快表未命中，则访问某个逻辑地址需要两次访问(注意：在找到页表项后，应同时将其存入快表，以便后面可

8.MMU和TLB

在CPU内部，有一个部件叫做MMU(内存管理单元)，由它来负责将虚拟地址映射为物理地址,MMU集成了TLB来存储CPU最近常用的页表项来加速寻址，TLB中找不到再去查询内存中的页表，可以认为TLB是MMU的缓存。

10.swap分区(交换技术)

10.1 swap分区

swap分区：它是硬盘中一块特殊的空间，多个进程在内存中并发运行，当内存空间紧张时，系统将内存中某些不常用的进程暂时换出外存(硬盘)，把外存中某些已具备运行条件的进程换入内存，操作系统为了保持对这些换出进程的管理，被换出进程的PCB(进程控制块)还是会留在内存当中。

swap分区优点：有了swap分区，通过操作系统的调度，应用程序实际可以使用的内存空间将远超系统的物理内存空间。

swap分区缺点：频繁地读写硬盘，会显著降低操作系统的运行速率。

10.2 交换技术

交换技术：内存空间紧张时，暂时把不用的进程由内存移动到外存，腾出内存空间，需要时再把相应进程全部调入内存的一种存储管理技术，因为换入换出是以整个进程为单位，所以进程大小受到实际内存的限制，一定要小于内存才行。

11.分页存储管理思想

把内存（物理内存）分为一个个相等的小分区，再按照分区大小把进程拆分成一个个小部分。这样内存空间就被划分成一个个大小相等的分区，每个分区就是一个“页框”或“内存块”，每个页框有一个编号，即“页框号”或“内存块号”，页框号从0开始的。将用户进程的地址空间（虚拟内存）也分为与页框大小相等的一个个区域称为“页”或“页面”，每个页面也有一个编号，即“页号”，页号也是从0开始的。操作系统以页框为单位为各个进程分配内存空间，进程的每个页面分别放入一个页框中，这样进程的页面与内存的页框形成了一一对应的关系。使用分页（Paging）的方式对虚拟地址空间和物理地址空间进行分割和映射，减小换入换出的粒度，提高程序运行效率

12.抖动

刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为称为抖动。产生抖动的主要原因是进程频繁访问的页面数高于可用的物理块数(分配给进程的物理块不够)。如果出现了抖动现象，系统会花大量时间去处理进程页面的换入换出，而实际用于进程执行的时间就变得的很少

13.用户态、内核态

内核态：内核空间存放的是操作系统内核代码和数据，它是被所有程序共享的，在程序中修改内核空间中的数据不仅会影响操作系统本身的稳定性，还影响其他程序，所以操作系统禁止用户程序直接访问内核空间。

用户态：用户空间保存的是应用程序的代码和数据，是程序私有的，其他程序一般无法访问。

1.内核和用户程序共用地址空间

让内核拥有完全独立的地址空间，就是让内核处于一个独立的进程中，这样每次进行系统调用都需要切换进程。切换进程的消耗是巨大的，不仅需要寄存器进栈出栈，还会使CPU中的数据缓存失效、MMU中的页表缓存失效，这将导致内存的访问在一段时间内相当低效。

而让内核和用户程序共享地址空间，发生系统调用时进行的是模式切换，模式切换仅仅需要寄存器进栈出栈，不会导致缓存失效；现代CPU也都提供了快速进出内核模式的指令，与进程切换比起来，效率大大提高了。

2.何时会从用户切换到内核

1.系统调用：用户进程通过系统调用申请操作系统提供的服务程序来完成工作。

2.发生异常：当CPU在执行运行在用户态的程序时，发生某些不可知的异常，就会触发由当前运行进程切换到处理这个异常的内核相关程序，也就到了内核态，比如**缺页异常**。

3.外围设备的中断

·当外围设备完成用户请求的操作之后，会向CPU发出相应的中断信号，这时CPU会暂停下一条要执行的指令，转去执行中断信号的处理程序，如果先执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了从用户态到内核态的切换。

操作系统：

- malloc底层实现原理
- 分配内存时什么时候会调用brk？空闲链表的分配和维护是在用户态完成还是内核态完成？
- 讲讲程序中的bss段
- 讲讲程序加载运行的全过程

内存池解决了什么问题